20 April 2016 CS1006 Practical 3 Student ID: 150012773

Wumpus

Overview

The third practical required from us to produce an implementation of a Hunt the Wumpus-style game. In order to complete the practical, a basic set of features were to be implemented: a working and rule-abiding copy of the game that could be payed by a single person; randomly generated pits and superbats; a text-based user interface the user could interact with.
In addition to the basic features the game we created implements: additional four directions of movement (NE, NW, SE, SW); clues for when bats are nearby; a graphical user interface, that in addition to displaying your current position, shows the available moves you have; an AI that avoids every room that poses a threat, looks for the treasure, and, when the treasure is found, tries to exit the cave in the safest way possible; a finite number of arrows, that can be retrieved by entering the room you have shot them in; four levels of difficulty for the game (Easy, Medium, Difficult, Expert), which change the size of the board, number of bats, and number of pits; 'null rooms' - random locations on the board, which allow the user to only move in certain randomly chosen directions - 'null rooms' open different 'doors' every time the player enters them.

Design

UI

The user interface directory contains three classes, these include the game panel, the information panel and the start panel. Each of these classes extend JPanel.
When the program is first run a new JFrame is created which houses the the start GUI panel. The panel allows the user to select the difficulty from a JCombo box, which by default is set to easy and then select whether they wish to play or spectate an AI playing. The different difficulties in the combo box represent a number; this number is used to generate the the size of the board and thus the harder the difficulty the larger the board generated. If the user chooses to play as player a new player object is created and the current location of that player is set to a random location on the board. The same processes happens when the user chooses to spectate an AI playing.

Once the selection is made this frame is disposed and a new frame containing the info panel and the game panel is created. In order to house both panels we have used a split pane. This frame is also invoked on a different thread so that the program can use a separate thread for just processing the GUI.

The info panel is used to keep track of the current status of the game. To create this panel we had to use a grid box layout, in order to add the labels and text area in the suitable positions.
The text area is used to display clues and the labels are used to display the the current game status e.g. win, lose or playing. The other labels display whether the wumpus is dead or alive, if you have the treasure, if you've shot and missed the wumpus and finally how many arrows you have.

Finally the game panel's main purpose is to house an array of buttons. When the game panel is initialized, an array of buttons is generated based on the size of the board. In order to keep all the buttons the same size, we used a grid layout. All the buttons have action listeners added to them upon creation in order for the program to tell which button has been clicked.

Upon initialisation the game panel takes in a game object so it can access the methods in the game class. It also takes in a boolean variable 'AI' which is used to check whether a player is actually playing or if the AI is playing. The reason for checking whether an AI is playing is because the buttons should not be able to be clicked if a player is spectating. Finally when the panel is created we have used Key bindings to map actions to specific keys to actions so when certain keys are pressed the program can execute certain actions.

Once the panel is created the game object is used to find the available moves for the player's current position. These available moves are represented by green buttons and the player's current location is represented by a red button. The GUI will only register clicks on green buttons, this stops the player from trying to move any location on the board.
So that the user has the option to shoot or move, we diced to implement a system, where the user must first select an available button then, if they wish to move, they must press the spacebar or if they wish to shoot they must press the enter key.
If the user decides to shoot the destination of the shot is sent to the board class and here the program checks whether the wumpus was hit. If it was missed, the wumpus' location is changed. The players arrows are then decremented appropriately .
If the user decides to move, the destination they select is also sent to the board class where the program checks if anything is in the destination room or sounding rooms.

Once the user has moved the GUI updates and displays the next set of available locations. If one of the available locations contains the treasure, the button containing the treasure is made orange. Similarly if one of the available locations contains the location of the exit , the button containing the exit is made purple. These highlightings then stay throughout the game unless the treasure is retrieved or the player has finished the game.

Finally if the user enters a room containing bats, a new random location is generated and the gui updates, with the player's new location. A red marker is then left in the player's original location to signify there are bats in that room.

Once the game is over, a pop up box is displayed which asks the user is they wish to restart.

AI

If the user wishes to play as an AI a new instance of the class called startAI is initiated. This class implements the 'runnable' interface. The reason for this is so that the processing done by AI can be done on a different thread, thus allowing the GUI to be updated simultaneously. The run method in this class contains a while loop which continuously makes the AI make a move until the game is over.

The AI package contains two classes, the first class is the AI player class which implements the IPlayer interface, this class is relatively simple and mainly contains getters and setters.

Upon initialization the AIplayer class creates a new Static Evaluation object. This object is used to determine which move the AI should make next. The static evaluation class first sees whether the AI has made a move yet, if it hasn't a random move is then made and sent to the board. If the AI has made its first move and now can make another move the static evaluation class is then used to determine which move is the best. This class forces the AI to play defensively and try to make safe moves whilst also attempting to explore new rooms.
In order to attempt to make the best move possible, the AI will first get all the available locations it can move to. It will then get all the previous available moves it could have made. A score is then assigned to the new room based on the number of warnings that room contains.
If there are no warnings in the new room, all the available rooms are added to a 'safeLocations' list, which contains all the locations which the AI knows does not contain any items. The AI will then check if it has already visited any of the available rooms. If there are rooms which it hasn't already visited but and are available, it will choose one. Otherwise a random room is selected.
If however the new room does contain a warning, the AI will check to see if the old room contained any warning and if so how many. If the AI's current location contains less warnings than the AI previous location, it will attempt to choose a room, that it safe and hasn't visited before, if there are no safe rooms available, it will choose a room which was not previously available to it.
Alternatively if the old room contained fewer warning than the room the AI just moved to, the AI will backtrack and then choose a different room to move to.

Logic

The logic package contains three classes.

Board

The board class contains methods that set up and update the state of the game. The getNewLocation() method is used to generate random integers in an array based on the size of the ArrayList locations used for initialising the board. These are used to place the Wumpus, pits, bats, treasure, and exit locations randomly. The check(int[] location) method returns a char that represents an item in the location handed in, for example if the location handed in contains the Wumpus the function will return a 'w' signifying there is a Wumpus in that location. If there is no item in the handed in location the function returns a space.
The returned character is used in the move and shoot methods. The move(int[] x, int[] y) method changes the currentLocation array coordinates of the player based on input. A different method from the Game class gets initialised based on the character returned from the check(int[] location) method. The player gets notified what the performed action was by getting a message from the Information panel that is created from the infoPanel class from the Interface package. The shoot(int[] x, int[] y) method checks the check(int[] location) method as well. Based on input, if the returned character from it is 'w', the position of the array in the ArrayList of the game gets changed from a Wumpus location to an Arrow location. If the returned character is a space, the integer array arrowLocations gets updated with that location and the wumpusLocation gets changed to a new random one in the ArrayList.The checkValidMoves() method adds a status to the infoPanel.getClues() method based on the character returned from the check(int[] location) method. The arrowEncounter(int[] location) method creates an ArrayList with all arrow locations, updates it based on their current placements and returns an integer for each room that has arrows depending on their number.

Player

This class implements methods from the interface IPlayer. The constructor sets the current location, number of arrows the player has, and sets the boolean tresureFound to false. Methods are primarily getters and setters. There are methods used for movement and shooting which override those in the Board class.

Game

The Game class constructor sets the board size, creates a board based on that size and sets a player (human or AI). Valid moves are set based on the defaultDirections for movement. Then, by using a for-loop, the findValidMovesForPos(int x, int y) method returns an ArrayList that gets all the possible moves based current location and the defaultDirections, and returns every available move. In order for the player to be able to wrap around, if-else checks have been added in order to make sure the move that is made is not out of bounds. The nullRoom() method creates a position in the ArrayList that has a variable number of possible moves randomly placed around the player's current position. Using a for loop, the properties of this position get changed every time it receives a player's current position on the board. The method validMove(int x, int y) returns a boolean and checks whether the location that the player chooses is a validMove. There are several methods returning booleans that check whether the current corresponds to a bat location, wumpus location, pit location, null rom location, treasure location or exit location.

Testing

Testing the game

Core functionality tests

Description	Expected Result	Actual Result	Comment
Pick difficulty of game	Set board size, number of bats and number of pits based on chosen difficulty.	Depending on difficulty sets board size, number of bats and number of pits.	Worked as expected
Pick player	Set player to AI or human player.	Sets player based on what you choose.	Worked as expected
Select available position	Position colour changes from green to blue and allows for an action to be made.	When you click on an available position, colour changes from green to blue and you are allowed to perform an action.	Worked as expected
Select unavailable position	Position colour does not change; action of any sorts cannot be performed.	When clicking on an unavailable position you are not allowed to perform any kind of action.	Worked as expected

Action: Move	Move to an already selected available position; change current location and update available locations.	When performing movement, board updates your new location and available locations.	Worked as expected
Action: Move Case: Arrow Encounter	Perform Action: Move; retrieve any arrows you have in your new current position.	When moving to a location you have already taken a shot in, you retrieve every arrow piece you have in there.	Worked as expected
Action: Move Case: Bat encounter	Perform Action: Move; update current place to a random new one on the board.	When moving to a location that has a bat in it, you get ‘teleported’ to a new random position on the board.	Worked as expected
Action: Move Case: Pit encounter	Perform Action: Move; lose game, resulting in an exit and terminating the program; ask for a restart.	When moving to a location that has a pit in it, a message that you lose the game appears. You are then asked whether you want to restart the game or not. If you do not wish to do so, the game terminates.	Worked as expected
Action: Move Case: Wumpus encounter	Perform Action: Move; 75% chance to lose the game; 25% chance for the Wumpus to go to another random position.	When moving to a location with a Wumpus in it, the monster kills you in 3 out of 4 cases. If you are left alive, the Wumpus changes his position.	Worked as expected
Action: Move Case: Null room encounter	Perform Action: Move; available locations change to a random number of previously available	When moving to a location that is a null room, you do not get the normal number of available	Worked as expected

	locations	locations; new available locations are randomly placed around you and are of a random number.	
Action: Move Case: Treasure encounter	Perform Action: Move; pick treasure; change coordinate state to empty; allow exit.	When moving to a treasure room, you get the treasure, the room state changes to empty, and you are allowed to exit.	Worked as expected
Action: Move Case: Exit encounter	Perform Action: Move; if treasure is picked win game, ask for restart afterwards; else get message that you need treasure.	When moving to an exit room if treasure is picked win game, ask for restart afterwards; else get message that you need treasure first.	Worked as expected
Action: Shoot	Shoot to a selected available position; if Wumpus is in room, it dies; if Wumpus isn't in room it changes position.	When performing shooting, your arrows decrease by one. If you have shot a Wumpus, it dies. Else the Wumpus changes position to another random position.	Worked as expected
Restart game	After either winning or losing a game, player is asked whether he/she would want to restart the game. If option 'yes' is chosen, you are asked to pick difficulty and choose player. If option 'no' is chosen, the program terminates.	When finishing a game by either winning or losing, the player is asked whether he/she would want to restart the game. If you chose 'yes', you are asked to pick difficulty and choose player. If option 'no' is chosen, the program terminates.	Worked as expected
Check clues	If you perform Action: Move, and	When moving to a position where one	Worked as expected

	either a Wumpus, Pit, Bat, Treasure of Exit is nearby, you get a specific clue which one of these is nearby.	of the available positions is not empty, a message that for the kind of ‘danger’ in the room appears.	
Perform action on an unavailable position	If you try to perform any kind of action, you are unable to do so.	When trying to shoot or move to a position which does not appear to be available, the action cannot be finalised.	Worked as expected
Re-entering null room	If you perform Action: Move, and you enter a location that is a null room you have entered before, available number of locations and their placement is different.	When moving to a null room that you have been to before, the number and placement of available locations around you updates to a different random one.	Works as expected
Bat location visual tracking	After entering a bat room and having been teleported to a new random location, the bat location will still appear on the map. However if you go near one such locations, it gets back to the normal grey colour.	When moved to a new random location by a bat, the locations of the bat stays visible on the map. Moving closer to the bat location resets the visual tracking	Works as expected
Treasure visual clue	After being near the treasure, you see its ‘glimmer’ and it appears on the map	When being near a treasure room, the location that the treasure is in appears in orange.	Works as expected

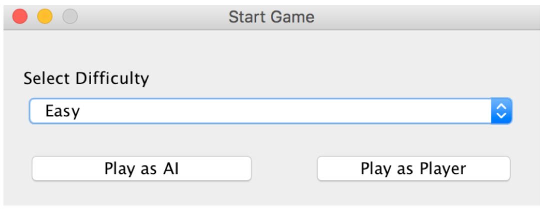
Exit visual tracking	If you have encountered the exit, but have not found the treasure, the exit location stays permanently on the map	When entering the exit location but not having the treasure, the room that the exit is in appears in magenta and stays permanently on the map	Works as expected

Testing the AI

After running 1000 instances of the game with the AI playing, the AI managed to win 22% of games. Although this is a somewhat disappointing outcome, the AI is still statically better than human players, whom on average only won 18% of games. This statistic is somewhat biased as we only tested the game with human players 50 times, moreover various people played the game.
In all the tests the game was completed and the AI did not run into any problems. The main issue is because there is a large number of bats and pit, this consequently means few rooms can be marked as safe and thus the AI will often have to make a random move if there are no better moves available.

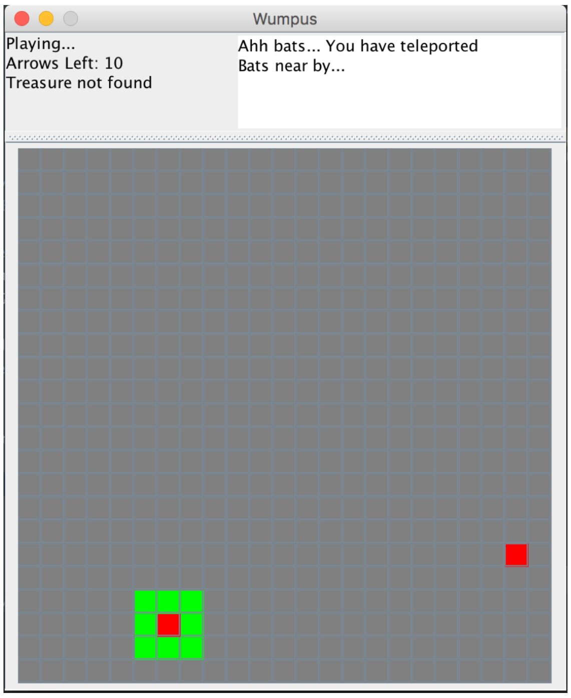
Sample OutPut

Main Menu



Select Difficulty: Easy

Board



Playing...
Arrows Left: 10
Treasure not found

Ahh bats... You have teleported
Bats near by...

Evaluation

The program we designed meets all the initial requirements successfully as well as incorporating various different extensions. After extensively testing the core functionality of the program, we are pleased to report that the program works as expected and is capable of handling erroneous conditions with ease.
The results from the AI were satisfying, although due to time constraints, we were unable have made the AI attempt to kill the Wumpus. Because we have used multiple threads the performance of the program is good.

Conclusion

To conclude we are happy with the solution we have created to the given problem. The code is efficient and neat, providing all the required basic functionalities, as well as a number of extensions.

Individual Contribution

We both worked together on the project and out individual contributions were fairly equally.